

$\mathcal{P} \neq \mathcal{NP}$ by Modus Tollens

Joonmo Kim

Applied Computer Engineering Department, Dankook University

South Korea

chixevolna@gmail.com

June 17, 2014

Abstract

An artificially designed Turing Machine algorithm \mathbf{M}^o generates the instances of the satisfiability problem, and check their satisfiability. Under the assumption $\mathcal{P} = \mathcal{NP}$, we show that \mathbf{M}^o has a certain property, which, without the assumption, \mathbf{M}^o does not have. This leads to $\mathcal{P} \neq \mathcal{NP}$ by modus tollens.

Notice Version 2 of this article, [2], is the final writing for the proof. Later versions are for adding meta data and replies. Version 5 is for adding comments for easier readings: subsubsection 0.1.1 and subsection 0.2. Version 6 is for modifying subsection 0.2.

1 Introduction

This formula $(P_1 \rightarrow (P_2 \rightarrow P_3)) \wedge (\neg(P_2 \rightarrow P_3))$ concludes $\neg P_1$ by modus tollens. As an example, we may easily prove that: it is false that each of all numbers can be described by a finite-length digits. Simple proof is somehow showing the number of infinite-length digits. Instead, a boy makes use of the modus tollens as follows. He lets the propositions be:

P_1 : each of all numbers can be described by a finite-length digits,

P_2 : there exists the irrational number,

P_3 : the irrational number can be described by a finite-length digits.

Then we know that $P_1 \rightarrow (P_2 \rightarrow P_3)$ can be proved, and so can $\neg(P_2 \rightarrow P_3)$.

In proving $\mathcal{P} \neq \mathcal{NP}$, we are going to follow his proof. Let P_1 be $\mathcal{P} = \mathcal{NP}$, P_2 an argument that an algorithm (say \mathbf{M}^o) exists, and P_3 an argument on the property of \mathbf{M}^o . It will be seen that if $\mathcal{P} = \mathcal{NP}$ then \mathbf{M}^o has the property of P_3 , otherwise it does not.

2 Algorithm M and Cook's Theory

Cook's Theory [1] says that the accepting computation of a non-deterministic poly-time Turing Machine on an input x can be transformed in a polynomial time to a satisfiable instance, denoted as \mathbf{c} , of the satisfiability problem(SAT).

All the conditions of being an accepting computation are expressed, in \mathbf{c} , as a collection of Boolean clauses. There are six groups of clauses in \mathbf{c} . Quoted from [1], each group imposes restrictions as:

G_1 : at each time i , Turing Machine M is in exactly one state,

G_2 : at each time i , the read-write head is scanning exactly one tape square,

G_3 : at each time i , each tape square contains exactly one symbol from Γ ,

G_4 : at time 0, the computation is in the initial configuration of its checking stage for input x ,

G_5 : by time $p(n)$, M has entered state q_y and hence has accepted x ,

G_6 : for each time i , $0 \leq i < p(n)$, the configuration of M at time $i+1$ follows by a single application of the transition function δ from the configuration at time i .

Though the groups of clauses are designed to represent the runs of poly-time Turing Machines, they can be modified to represent the runs of longer-time ones. As long as there exists any finite-length accepting computation path from a problem instance to an accepting state over a Turing Machine, how long the path may be, there may exist the corresponding clauses for each of all the transitions along the path.

Therefore, we may extend the meaning of an accepting computation as the representation of a finite run of a Turing Machine on one of its accepting input, regardless of the run time. Note that, in [1], the time for the transformation from an accepting computation to the corresponding SAT instance should have been kept within a polynomial, but the proof here does not regard any kind of run time, as long as the run is finite.

We may observe that the clauses in \mathbf{c} can be divided into two parts: one is for the representation of the given input, and the other the run of the Turing Machine on the input. Let the *input-part* be the clauses for the description of the input x , which is denoted as \mathbf{c}^x . Let the *run-part* (denoted as \mathbf{c}^r) be the part of \mathbf{c} such that the clauses for \mathbf{c}^x are cut off from \mathbf{c} , i.e., \mathbf{c}^r is a part of \mathbf{c} that actually represents the operations of the corresponding Turing Machine (grouped in [1] as G_1, G_2, G_3, G_5, G_6).

Let \mathbf{M} be a Turing Machine algorithm, to which the input is a string denoted by y . \mathbf{M} is designed to include a finite number of \mathbf{c}^r 's (i.e., $\mathbf{c}_1^r, \dots, \mathbf{c}_n^r$), which are trimmed from \mathbf{c} 's (i.e., $\mathbf{c}_1, \dots, \mathbf{c}_n$), where \mathbf{c} 's are arbitrarily selected accepting computations. That is, \mathbf{c}_i^r is formed by cutting off \mathbf{c}_i^x from \mathbf{c}_i , where x is the input (represented by the initial configuration in G_4) of the computation that corresponds to \mathbf{c}_i . According to the run-parts included, countably many \mathbf{M} 's can be constructed, and they can be somehow ordered as: $\mathbf{M}_1, \mathbf{M}_2, \dots, \mathbf{M}_i, \dots$. For \mathbf{M}_i , its run-parts can be written as: $\mathbf{c}_{i1}^r, \mathbf{c}_{i2}^r, \dots, \mathbf{c}_{im}^r$.

Given an input y , during the run of \mathbf{M}_i , \mathbf{c}^y and \mathbf{c}_{ij}^r are concatenated, forming \mathbf{c}_{ij} ($1 \leq j \leq m$). For each \mathbf{c}_{ij} , the module of SAT-solver, which will accordingly

be chosen to be either a deterministic algorithm or non-deterministic, performs the run for the satisfiability check. If \mathbf{c}_{ij} is satisfiable then \mathbf{M}_i increases its counter, and goes on to the next \mathbf{c} ; this process repeats up to \mathbf{c}_{im} . At the end of the counting, \mathbf{M}_i accept y if the counter holds an odd number. That is, the task of \mathbf{M} , at the given input of a finite string y , is to count the number of satisfiable \mathbf{c} 's, and accept y if the counted number is odd. Observe that it is not impossible for \mathbf{c}_{ij} to be satisfiable though the chances are usually rare, and that the run time of \mathbf{M}_i shall be finite because each of all the satisfiability check will take a finite time. These observations ensure that \mathbf{M}_i determines a set of acceptable strings, supporting \mathbf{M}_i to be a Turing Machine.

3 Algorithm \mathbf{M}^o and the Theorem

Let a *particular transition table* of a Turing Machine be a transition table particularly for just one or two accepting problem instances, where the computation time does not matter as long as it is finite. So, a particular transition table for an accepting problem instance may produce an accepting computation by running on a Turing Machine. Observe that each of all accepting computations may have its particular transition table, i.e., the table can be built by collecting all the distinguished transitions from the computation, where we know that a computation is a sequence of the transitions of configurations of a Turing Machine.

We then introduce an algorithm, denoted as \mathbf{M}^o , which is one of \mathbf{M} 's with the property as follows. Let $\hat{ac}_{\mathbf{M}}$ be the accepting computation of the run of \mathbf{M} on an input y , let \mathbf{t} be a particular transition table for $\hat{ac}_{\mathbf{M}}$, let \mathbf{c}^o be one of \mathbf{c} 's that appear during the run of \mathbf{M} , and let $\hat{ac}_{\mathbf{c}^o}$ be the accepting computation, which is described by the clauses of \mathbf{c}^o . If \mathbf{t} is also a particular transition table for $\hat{ac}_{\mathbf{c}^o}$ then denote \mathbf{M} as \mathbf{M}^o .

For the run of \mathbf{M}^o , we may consider two types of particular transition tables. Let D_{sat} be the particular transition table, by which \mathbf{M}^o runs deterministically and the SAT-solver module runs deterministically in a poly-time for the length of \mathbf{c} . Analogously, we may have ND_{sat} , by which \mathbf{M}^o runs non-deterministically and the SAT-solver module runs non-deterministically in a poly-time for the length of \mathbf{c} .

Theorem 1. $\mathcal{P} \neq \mathcal{NP}$

Proof. Let P_1 , P_2 and P_3 be the following propositions:

P_1 : $\mathcal{P} = \mathcal{NP}$,

P_2 : \mathbf{M}^o exists,

P_3 : there exists \mathbf{t} , which is D_{sat} .

By modus tollens, $(P_1 \rightarrow (P_2 \rightarrow P_3)) \wedge (\neg(P_2 \rightarrow P_3))$ may conclude $\neg P_1$.

1) Proposition $P_1 \rightarrow (P_2 \rightarrow P_3)$ is to show that if \mathbf{M}^o exists then there exists \mathbf{t} , which is D_{sat} , when $\mathcal{P} = \mathcal{NP}$ is the antecedent.

By $\mathcal{P} = \mathcal{NP}$, there exists a deterministic poly-time SAT-solver, so the SAT-solver module in \mathbf{M}^o can be implemented to be a deterministic algorithm, which

runs in a poly-time for the length of \mathbf{c} . Thus, \mathbf{t} can be D_{sat} .

2) For the latter part, we are to show $\neg(P_2 \rightarrow P_3)$. By the inference rule, if P_2 is true then a contradiction from $P_2 \rightarrow P_3$ implies $\neg(P_2 \rightarrow P_3)$.

We can show that P_2 is true, as follows. For any chosen \mathbf{c}^o , build two non-deterministic particular transition tables for $\hat{ac}_{\mathbf{M}^o}$ and $\hat{ac}_{\mathbf{c}^o}$ separately, and then merge the two so that one of the two computations can be chosen selectively from the starting state during the run. \mathbf{M}^o may exist by this \mathbf{t} , which is ND_{sat} .

Next we are to derive a contradiction from $P_2 \rightarrow P_3$. Observe that $P_2 \rightarrow P_3$ implies specifically this *argument*: if \mathbf{M}^o exists then there exists \mathbf{t} , which is D_{sat} particular transition table for both $\hat{ac}_{\mathbf{M}^o}$ and $\hat{ac}_{\mathbf{c}^o}$. Then, by the *argument* together with the definition (of \mathbf{M}^o) that $\hat{ac}_{\mathbf{M}^o}$ and $\hat{ac}_{\mathbf{c}^o}$ are the accepting computations that share the same input, it is concluded that both $\hat{ac}_{\mathbf{M}^o}$ and $\hat{ac}_{\mathbf{c}^o}$ are exactly the same computation, i.e., all the transitions of the configurations of $\hat{ac}_{\mathbf{M}^o}$ and those of $\hat{ac}_{\mathbf{c}^o}$ are exactly the same.

According to [1], with the assignment of the truth-values, which are acquired by the SAT-solver module, to the clauses of \mathbf{c}^o , we later may acquire $\hat{ac}_{\mathbf{c}^o}$ in the form of the sequence of transitions of configuration of a Turing Machine.

Now, let i be the number of the transitions between the configurations in $\hat{ac}_{\mathbf{M}^o}$, j the number of the clauses of \mathbf{c}^o , and k the number of the transitions between the configurations in $\hat{ac}_{\mathbf{c}^o}$.

Since, at the least, all the clauses of \mathbf{c}^o should once be loaded on the tape of the Turing Machine as well as other \mathbf{c} 's, we may have $i > j$, and since it is addressed in [1] that each transition of an accepting computation is described by more than one clauses, we may have $j > k$, resulting $i > j > k$.

However, $i = k$ because it is concluded above that both $\hat{ac}_{\mathbf{M}^o}$ and $\hat{ac}_{\mathbf{c}^o}$ are exactly the same computation. This is a contradiction from $(P_2 \rightarrow P_3)$. \square

As a commentary, to make sure that the proof does not fall into the similar oddity of the well-known relativizations of \mathcal{P} vs. \mathcal{NP} , it'd be better to consider this argument:

$(Q_1 \rightarrow (Q_2 \rightarrow Q_3)) \wedge (\neg(Q_2 \rightarrow Q_3))$, where

Q_1 : $\mathcal{P} \neq \mathcal{NP}$,

Q_2 : \mathbf{M}^o exists,

Q_3 : there exists \mathbf{t} , which is ND_{sat} .

Similarly, by the antecedent $\mathcal{P} \neq \mathcal{NP}$, there exist only non-deterministic algorithm for the SAT-solver module, so $Q_1 \rightarrow (Q_2 \rightarrow Q_3)$ can analogously be proved. For the latter part, $\neg(Q_2 \rightarrow Q_3)$ become true if $Q_2 \rightarrow Q_3$ implies $i = k$, as in the proof. If so, we have the result that $\mathcal{P} = \mathcal{NP}$.

However, $Q_2 \rightarrow Q_3$ may imply $i = k$ if $\hat{ac}_{\mathbf{M}^o}$ and $\hat{ac}_{\mathbf{c}^o}$ are the same, but we know, referring to the proof, that there is no such \mathbf{t} that makes $\hat{ac}_{\mathbf{M}^o}$ and $\hat{ac}_{\mathbf{c}^o}$ the same. In fact, we may build many \mathbf{t} 's that does not incur $i = k$ from $Q_2 \rightarrow Q_3$, as mentioned in the proof.

Replies to the Critiques

Followings are the replies to the critiques and comments for the proof. Replies to future critiques will be added below. Author does not expect the success of the proof, rather he is waiting to see what is wrong in [2].

0.1 Replies to the Critique of J. Kim’s “P is not equal to NP by Modus Tollens”

It is quite likely that the Critique of [3] has errors. Authors of [3] has a good understanding on [2], but missed some points.

The tables in 2.4 of [3] are exactly what the author of [2] expected. (Thanks.)

“3.1 Invalidity of logical argument” of [3] said that ‘ \mathbf{t} is D_{sat} ’ is a fact, but \mathbf{t} can also be ND_{sat} especially when the SAT-solver module is implemented non-deterministically. Notice that the TM algorithms designed for the proof are not practical programming. As a result, ‘ \mathbf{t} is D_{sat} ’ is not always true. In addition, \mathbf{M}^o may exist when \mathbf{t} is ND_{sat} .

For 3.2 of [3], the author of [2] would rather choose “3.2.2 Second interpretation.” The critique mentioned : “Note that these accepting computations are not necessarily the same as the accepting computations produced by their respective Turing machines’ transition tables.”

Author of [2] agree with this, but, for the proof of [2], it is enough if there exist more than one **cases** that *the accepting computations are the same as the accepting computations produced by their respective Turing machines transition tables*. The proof derives the contradiction from one of the **cases**. The **case** can be seen by the merged table in 2.4.

In 3.3, the critique said “At the end of his paper, Kim verifies that . . .” The part, ‘*the end of his paper,*’ is to show that the proof will not be fallen into the relativizations of \mathcal{P} vs. \mathcal{NP} . The author of [2] wanted to show that if the proof is correct then there is no room for the proof to be fallen into the relativization matters. Perhaps, this part can be omitted for now, while questioning on the correctness of the proof itself.

Comments Authors of [3] show a good understanding on [2], but have the errors as above that seem to imply that [2] has not yet been refuted.

0.1.1 Complementary to Subsection 0.1

Considering the view of the the critique of [3], author of [2] would like to add more explanation to guide other readers for easier and quicker understanding.

First of all, author of [2] does not think that the proof of Theorem 1 in [2] has to be divided into two interpretations: *3.2.1 First interpretation* and *3.2.2 Second interpretation* as in [3].

It seems that authors of [3] had the two interpretations because they think that \mathbf{t} and the two accepting computations, $\widehat{ac}_{\mathbf{M}}$ and $\widehat{ac}_{\mathbf{c}^o}$, should be derived from their respective Turing machines' transition tables so that the property $i > j > k$ can be applied correctly, to ensure a consistent and coherent frame for the proof, in the proof of Theorem 1.

However, such a frame does not let the proof of Theorem 1 reach the completion: this is what authors of [3] claim.

If the proof of Theorem 1 is correct, author of [2] is claiming a larger frame, in which a consistent and coherent proof can be ensured.

We see in the proof that there are two ways that \mathbf{t} 's are made. Firstly, in part 1) of the proof, \mathbf{t} , which is D_{sat} , comes into existence since there exist the respective Turing machine and its transition table. In fact, this \mathbf{t} is a special case of the transition table of the respective Turing machine, \mathbf{M}^o . As mentioned, \mathbf{M} , which is the base of \mathbf{M}^o , is composed of the SAT-solver module and the remaining part, where the module and the part can be implemented either way deterministically or non-deterministically. Actually, since the remaining part of \mathbf{M} can be programmed simply, it can easily be implemented deterministically. Thus, by the antecedent P_1 , all parts of \mathbf{M}^o can be implemented deterministically, resulting the existence of \mathbf{t} , which is D_{sat} . The second of making \mathbf{t} is the way that the tables are merged, which is rephrased in 2.4 of [3]. So, this \mathbf{t} is not related to the respective Turing machine.

By the way, the property $i > j > k$ is the one for the relationship between the accepting computations, $\widehat{ac}_{\mathbf{M}^o}$ and $\widehat{ac}_{\mathbf{c}^o}$, in \mathbf{M}^o : nothing about \mathbf{t} is considered in $i > j > k$. Therefore, how \mathbf{t} is made does not matter in the proof, just the existence/non-existence of \mathbf{t} for \mathbf{M}^o is important. This dispenses with the considerations of the two interpretations of [3].

0.2 Extension of the Proof: Why failed so far

If the proof of Theorem 1 in [2] is correct, probably we may see why the lower-bound, which is expected to be a deterministic exponential/super-polynomial time, of NP-Complete problems has not yet been found.

By substituting P_1 as “there exist deterministic exponential/super-polynomial time Turing Machines that solves SAT” and modifying the definition of D_{sat} to be “the particular transition table, by which \mathbf{M}^o runs deterministically and the SAT-solver module runs deterministically in an *exponential/super-polynomial* time for the length of \mathbf{c} ,” Theorem 1 can be analogously extended to claim that *there does not exist any deterministic exponential/super-polynomial time Turing Machine that solves SAT*.

Then, what about the fact that the brute-force search solves SAT? We may conclude that the brute-force search is not equal to the deterministic exponential/super-polynomial time Turing Machine algorithm, as follows.

Though the brute-force search for SAT problem can be implemented by a deterministic transition table that causes an exponential time, the run by the table on an input is a non-deterministic computing because the run is a sequence of generating(guessing) candidate solutions and checking them. Since the corresponding computing is non-deterministic, the particular transition table for the brute-force search is inherently non-deterministic.

So, letting P_1 be “there exist a brute-force search that solves SAT,” $P_1 \rightarrow (P_2 \rightarrow P_3)$ in the proof is not true, while it is true when P_1 is as above in this section.

References

- [1] M. R. GAREY and D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of \mathcal{NP} -Completeness*, W. H. Freeman & Co., New York (1979).
- [2] Joonmo Kim. P is not equal to NP by Modus Tollens. arXiv.org, CoRR, 2014. <http://arxiv.org/abs/1403.4143v2>.
- [3] Dan Hassin, Adam Scrivener, and Yibo Zhou. Critique of J. Kim’s “P is not equal to NP by Modus Tollens”. arXiv.org, CoRR, 2014. <http://arxiv.org/abs/1404.5352>.